Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation

Bruce R. Childers

University of Pittsburgh
Pittsburgh, Pennsylvania, USA
childers@cs.pitt.edu
http://www.cs.pitt.edu/coco

This work is a collaboration of many people!
Sponsored by Next Generation Systems, National Science Foundation.

Code Optimization

- Sophisticated algorithms exist for many optimizations that do quite well
- We are at the point of diminishing returns in applying optimizations - small gains are considered good
- The challenge is to go beyond current optimization improvements

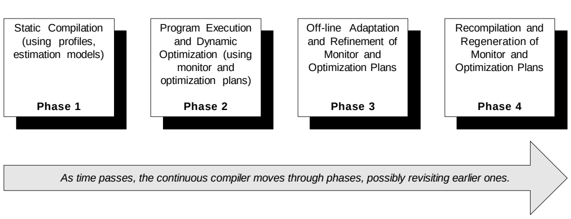
Continuous Compilation

- A new approach: Apply optimizations both statically at compile-time and dynamically at run-time with static planning
- Plan for both static and dynamic optimizations
  - Understand interactions of existing optimizations
  - Efficacy of both static and dynamic optimizations
- Determine what optimizations to apply, where to apply them, the order in which to apply them, and their parameters

Outline

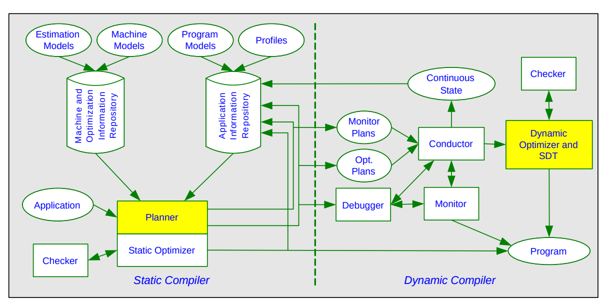
- Introduction: Continuous Compilation
- CoCo Framework
- Profit-driven Optimization
  - Loop optimizations
  - Scalar optimizations
- CoCo Run-time System
  - Software dynamic translation
  - Program instrumentation and optimization
- Summary

Continuous Compilation

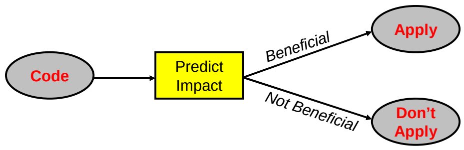


Target applications: Long running programs that have different phases of execution.

CoCo Framework



CoCo's Preliminary Prediction Framework



- Predict the impact of optimizations
- With estimates of benefit and penalty:
  - Decision about what optimizations to apply
  - Where to apply them
  - In what order to apply

Taking into account the code context and machine resources

Motivation

- Performance problems of the optimizations
  - Optimizations may degrade performance in some circumstances
  - Optimizations interact with one another by enabling and disabling other optimizations
- Optimizations often applied in an ad hoc fashion
  - Simple heuristic: always apply if applicable
  - Predetermined order to apply optimizations
  - Fixed configurations of optimizations

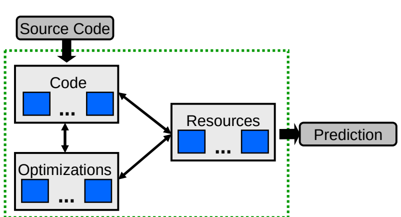
Challenges

- Performance varies widely, based on
  - Code context (e.g., loop trip count)
  - Configuration of optimizations (e.g. loop unrolling factor)
  - Machine configurations (e.g. cache configuration)
  - The order of optimizations
- Many resources impact overall performance
  - Cache configuration
  - Instruction scheduling rules
  - Register numbers and types

Our Approach

- Build and develop analytic models to predict when to apply an optimization, without actually applying the optimization
  - Need models of particular optimizations
  - Need models of the code
  - Need models of the resources that are effected
- Based on analytic models, make decisions about what optimizations to apply
  - We don't need accurate models, just the trend needs to be accurate enough to do the estimates

Framework for Predicting Optimizations

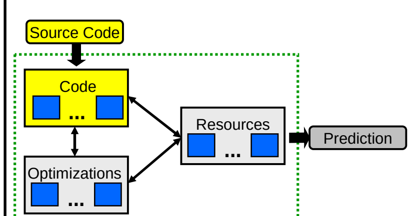


Plug 'n Play Models
1. Extract code context
2. Model effect of the optimization
3. Model effect on the machine resources

FPO: a Framework, consisting of models, for Predicting the impact of Optimizations

Consider both loop and scalar optimizations (e.g., PRE)

Loop Code Model



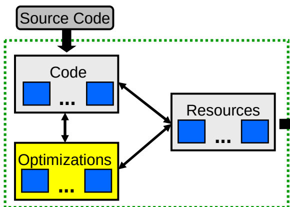
$$loop = \oint_{lb}^{ub} step\langle Refs\rangle$$

$$Refs = A \times I + C$$

Express code characteristics that effect resources

For loops and cache: 1. loop header, 2. array references, 3. reference sequence

Loop Optimization Model



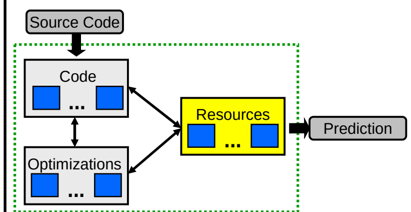
E.g., Interchange

$$f_{inter}(\oint\oint\ldots\oint g(\langle R\rangle))$$
$$g(\langle R\rangle)=\langle\forall(r\in\langle R\rangle)h(r)\rangle$$
$$h(r)=(l(A),C)$$
$$l(A)=A[:][i]\leftrightarrow A[:][j]$$

Model how optimization transforms code model

For loops and cache: sequence of functions that effect aspects of the loop's representation (code model)

Loop Resource Model (Cache)



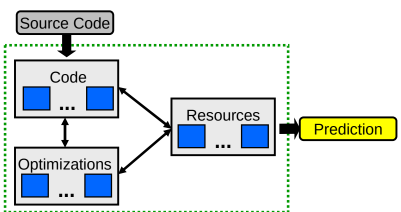
1. Group references by temporal or spatial reuse and compute conflicts
2. Pick representative ref from each group
3. Compute misses for each representative by reuse and conflicts

Simplified Ghosh model

How the code model effects machine resources

For loops and cache: with code model, how the array reference pattern effects both cache misses and hits

Loop Optimization Prediction

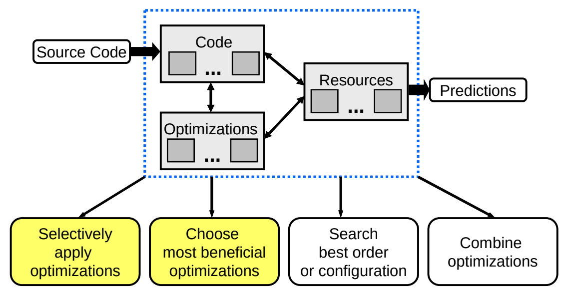


Prediction of the benefit or penalty is the difference between unoptimized and optimized

Compare transformed & non-transformed code

For loops and cache: Before & after loop optimization and whether optimization had reduction/increase in cache misses

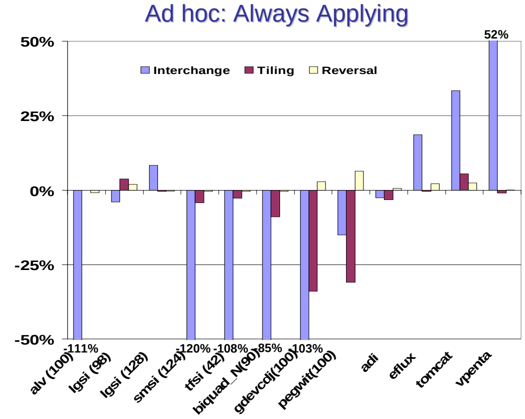
Applications of FPO



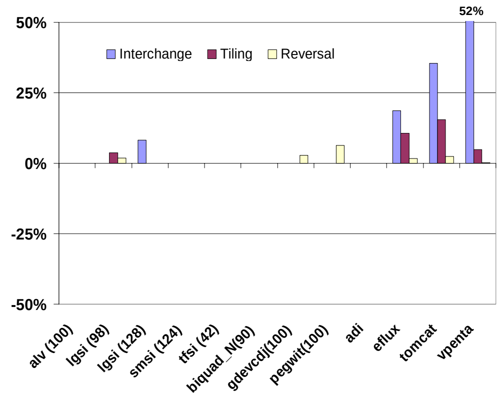
Experiments

- Benchmarks (MediaBench, DSPStone, and others)
- Machine model
  - SimpleScalar microarchitecture simulator
  - In-order, single issue, non-blocking cache
    - » 1KB cache, direct mapped and 32B line size
  - Similar to ARM's 94x, IBM Power PC 405
- Usefulness of FPO for loop optimizations
- Prediction accuracy of FPO for loop optimizations

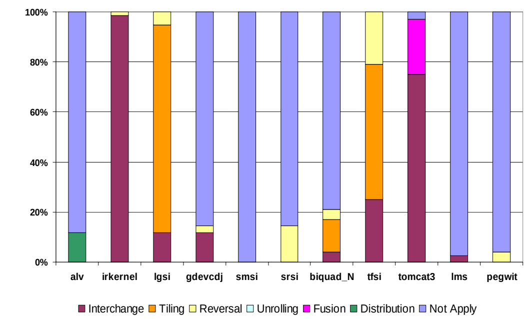
Ad hoc: Always Applying



Selectively Applying Loop Optimizations



Choose Most Beneficial Loop Optimization



Loop Opt. Prediction Accuracy

- Accurate trend
  - Whether an optimization is beneficial or not
- Correct prediction
  - When prediction matches actual execution behavior
- Prediction accuracy
  - Single loop nest: varying trip count
  - Multiple loop nests: the number of loop nests

Loop Opt. Prediction Accuracy

Prediction accuracy for single loop nest

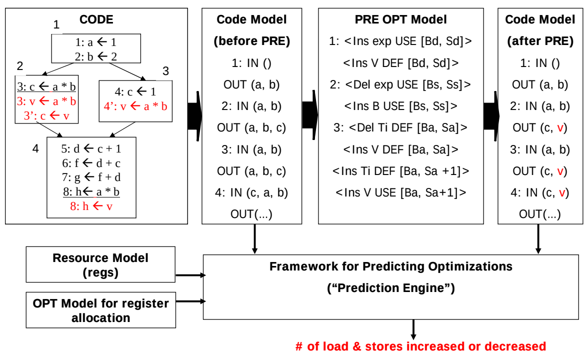
Benchmark	Interchange	Tiling	Reversal
alv	100%	100%	97.4%
irkernel	98.7%	100%	93.4%
lgsi	100%	100%	82%
smsi	100%	100%	86.8%
srsi	100%	100%	86.8%
tfsi	100%	97.4%	100%
tomcat3	98.7%	92.1%	93.4%
biquad_N	89.5%	88.2%	100%
gdevcdj	100%	100%	97.4%
lms	97.4%	100%	94.7%
pegwit	100%	100%	81.6%
Average	99%	98%	92%

Similar results for multiple loop nests

FPO: Scalar Optimizations

- Transformations that operate on scalar code
  - E.g., constant propogation, dead code elimination, partial redundancy elimination
- Can have several impacts
  - Reduce amount of computation
  - Change register pressure (for the better *or* for the worse!)
  - May change memory referencing pattern and cache behavior
- FPO (initially) considers
  - Affect on computation
  - How register pressure helps or hurts spills and reloads

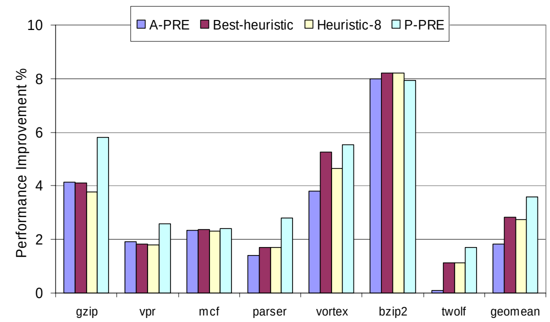
Impact of PRE



Using a Heuristic to Make Decisions

Bench.	Heuristic-driven PRE				Heuristic-driven LCIM			
	H 0	**H 4**	**H 8**	**H 16**	**H 0**	**H 4**	**H 8**	**H 16**
gzip	3.50	3.75	3.78	**4.10**	2.90	3.29	**5.40**	3.27
vpr	1.22	0.75	1.81	**1.83**	-0.40	-0.38	0.52	**0.69**
mcf	**2.37**	2.35	2.31	2.22	2.50	**2.62**	2.58	2.47
parser	1.25	1.50	**1.70**	1.35	2.55	**2.86**	1.99	2.23
vortex	4.73	**5.25**	4.66	3.86	4.88	**5.69**	4.99	5.28
bzip2	7.35	7.52	**8.19**	7.91	7.02	**7.35**	6.70	4.57
twolf	1.07	0.88	**1.14**	0.02	0.52	0.38	**2.14**	1.91

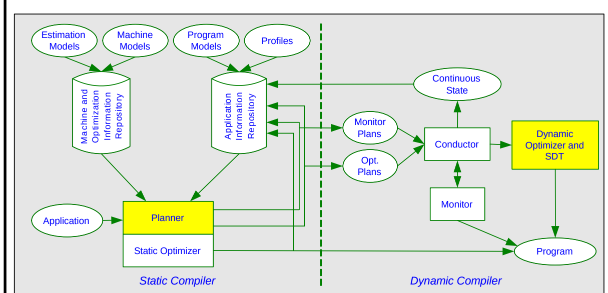
Approaches for PRE



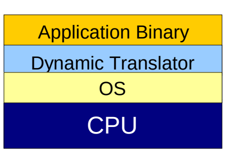
Scalar Opt. Prediction Accuracy

Benchmark	PREs	Correct	%Accuracy
gzip	48	43	89.58
vpr	303	291	96.04
mcf	51	44	86.27
parser	293	210	87.87
vortex	530	431	81.13
bzip2	56	44	78.57
twolf	475	433	91.12
Average			87.23

CoCo Framework

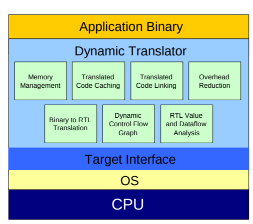


CoCo Run-Time System



- Based on *Software Dynamic Translation*
- Layer of software between application binary and the OS/CPU.
- Application's instructions are examined and modified before being executed on the CPU.
- Uses include binary translation, dynamic optimization, & others

CoCo Run-Time System



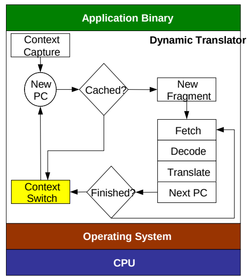
- Strata SDT developed at UVA & Pitt
- Provides basic functionality for run-time optimization
  - Memory management
  - Caching
  - Code analysis
  - **Overhead reduction**
  - **Program instrumentation**
- Includes target Interface
  - Handles all interactions between VM, application binary, and target OS/CPU

Current targets: MIPS/Irix, SPARC/Solaris, x86/Linux, MIPS/Playstation 2

Software Dynamic Translation

Translation and Execution

1. Fetch gets next instruction
2. Decode classifies instruction
3. Translate performs any necessary modification and rewriting and write translated instruction into fragment cache
4. Terminate fragment on control transfer and execute fragment

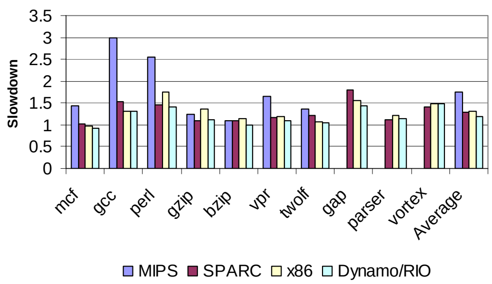


Strata Software Dynamic Translation

Overhead Reduction in Strata

- Improve run-time performance of translated code, *without* applying code optimizations
- Efficient execution in fragment cache
  - Linking branches in the translated cache
  - Handling indirect branches inline in the translated code
- Program instrumentation
  - Reduce amount of instrumentation inserted
  - Reduce the cost of an individual piece of instrumentation code

Strata Performance



No optimizations applied – basic run-time system overhead

Program Instrumentation

- CoCo monitors for run-time conditions
  - When conditions are found: Apply some optimization plan
- Program monitoring needs instrumentation
  - Instrumentation code is inserted into translated binary
  - Gathers information, may invoke some action
  - Event-driven model
- Challenges
  - Dynamic instrumentation: Insert and remove at run-time
  - Portable mechanisms (i.e., retargetability)
  - **Low run-time overhead** (including the insertion & removal)

Approach: Instrumentation Optimization

- Costs associated instrumentation
  - Probe count: Number of probes executed
  - Probe cost: Intercept program execution
- Transform instrumentation to reduce costs
  - Dynamic probe coalescing: Reduce probe count
  - Partial context switch: Reduce probe cost
  - Payload partial inlining: Reduce probe cost for hot code
- Applied for static or dynamic instrumentation

Example: Cache Simulation

- Cache simulation can be slooooow
- Direct-execution cache simulators
  - Popular (& compiled simulation) – Shade, Embra, Fast-Sim
  - Instrument every load and store – call data cache simulator
  - Instrument every basic block – call instruction cache simulator
- Used INSOP and Strata to build a very fast direct-execution simulator

Comparison of Cache Simulators

Benchmarks	Native (Secs.)	Shade	Sim-cache	Strata-Embra	Strata-Embra-O6
mcf	1,813	10.3x	23.4x	5.5x	2.5x
twolf	3,534	74.9x	N/A	15.7x	7.2x
gcc	1,364	111.1x	112.4x	25.2x	11.1x
vpr	831	113x	109.8x	20.4x	10.1x
parser	1,979	48.4x	123.2x	21.6x	8.8x
vortex	2,747	101.9x	114.4x	21.5x	10.6x
gzip	1,192	228.1x	227.9x	34x	13.1x
bzip	1,325	169.2x	194.1x	26.7x	8.1x

Average improvement over Strata-Embra is 2.4x

Related Work

- Effective optimization
  - Adaptive optimizing compilers: [Cooper02,04 & Whalley04]
  - Iterative compilation: [Knijnenburg03]
  - Optimization space exploration: [Triantafyllis03]
  - Analytic models: [Wolf91, Sarkar 97, McKinley96, Hu02]
- Software dynamic translation
  - Dynamo/RIO, Dynamo, Mojo, Vulcan, Walkabout, DELI
- Cache simulation
  - Shade, Embra, Fast-Cache, FastSim

Summary

- A new planning-based approach to compilation called Continuous Compilation (CoCo)
- Apply whole suite of optimizations with constant refinement of optimizations and plans for them
- Results
  - Highly accurate predictions for simple **loop optimizations**
  - Highly accurate predictions for **scalar optimizations**
  - **Low overhead run-time system** based on SDT
  - INSOP **reduces instrumentation cost** (cache simulation)

Collaborators

Many students have participated, including:

- FPO: Min Zhao (Pitt)
- Program instrumentation: Naveen Kumar (Pitt)
- Strata: Kevin Scott (UVA/Google) and Naveen Kumar
- Overhead reduction: Kevin Scott, Naveen Kumar, Jason Mars (Pitt)

Other Faculty: Mary Lou Soffa, Jack Davidson (UVA)

- Sponsored by the National Science Foundation, Next Generation Systems, 2002-2003 and 2003-2006

Current Areas of Focus

- Continuous compilation
- Software dynamic translation
  - Low overhead dynamic translation
  - New applications to architecture simulation, security
- Debugging of dynamically translated code
  - Dynamically optimized code
  - Security checking
  - Dynamically compiled simulation
- Soft error detection & recovery based on SDT
- Power-aware memory systems

Current Projects

- **Debugging dyn. translated/optimized code** – N. Kumar
- **On-demand structural software testing with dynamic instrumentation** – J. Misurda and J. Clause
- **Static/dynamic optimization planning** – S. Zhou
- **Optimization checking** – Y. Huang
- **Compiler-driven power management** – N. Aboughazaleh
- **Memory systems for cognitive processing**
- **Reuse through Speculation on Traces** – M. Pilla (from UFRGS)

Selected Past Projects

- **Instruction code compression/decompression**
- **On-demand code downloading for Smartcards**
- **Program profiling primitives and profiling language**
- **Software based value reuse on traces**

- **Power on/Shut down of superscalar functional units**
- **Memory bus reordering for power reduction**
- **Processor-driven DVS (based on IPC/peak demands)**
- **Data width-sensitive VLIWs & scheduling**
- **Application-specific processors (automatic design and target architectures)**

CS2002 Projects?

- Debugging for code security with SDT
- Dynamically compiled & sampled architecture simulation
- Profit-driven optimization for other constraints (e.g., power, code size)
- Self-checking programs (for soft errors; e.g., memory bit flips)
- Domain specific languages for structural testing and automatic planning
- Reconfigurable / custom memory systems
- And many others… or your ideas??

Let’s Talk!

- 6409 Sennott Square
- Office hours: MW 1-3 PM
- Or by appointment

- Send e-mail…. **childers@cs.pitt.edu**

- See selected papers online….
  http://www.cs.pitt.edu/~childers